On Evaluating Parallel Computer Systems

George B. Adams III
Robert L. Brown
Peter J. Denning

September 1985

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 85.3

RIACS

Research Institute for Advanced Computer Science

On Evaluating Parallel Computer Systems

George B. Adams III
Robert L. Brown
Peter J. Denning

Research Institute for Advanced Computer Science

RIACS TR 85.3
September 1985

Prepared by RIACS under NASA Contract No. NAS 2-11530 and DARPA Contract No. BDM-S500-0X6000. The content of this document does not represent the official position of NASA or DARPA.

---

1. Introduction

Modern supercomputers are within a factor of 10 of the 1 GFLOPS (approximate) speed limit for any single-processor architecture. By 1990 scientific and engineering applications will require speeds well beyond this limit. Computer architectures capable of massive parallelism are necessary to achieve such speeds. Because these machines are based on models of computation different from the familiar sequential process model, intuition is not a reliable basis for predicting how easy the new machines will be to program or how fast they are capable of running new algorithms. Instead, a systematic experimental approach is needed.

Because the architectures most capable of massive parallelism are also most different from anything familiar, there are no real applications programs to use as benchmarks, there are no users familiar with the applicable programming languages, and there are few working prototypes on which to run experiments. Because of the long lead time and great expense to develop a radically new architecture, we can ill afford to defer all serious attempts at evaluation until the machine has been delivered.

This report explores principles to guide the evaluation of computer systems employing new architectures. In summary, they are:

1. Evaluation must be an integral, ongoing part of a project to develop a computer of radically new architecture.
2. The evaluation process should seek to measure the usability of the system as well as its performance.
3. Users from the application domains must be an integral part of the evaluation process.
4. Evaluation results should be fed back into the design process.

The inspiration for these principles comes from three main sources. The first is a trend noticeable in major NASA projects. More and more NASA missions incorporate the concept that the resulting systems will be national resources for industry and science. Notable examples include the NAS (Numerical Aerodynamic Simulator), a complex of supercomputers for research in computational fluid dynamics; the Space Shuttle, for scientific experiments and satellite deployments; and the Space Station, for scientific research and space-borne manufacturing. These systems are very expensive and take many years to mature. Because of their complexity, it is impossible to forecast the details of how they will be used. NASA therefore seeks ways to involve users in the development of these systems so that the modes of human-machine interaction can be studied as they evolve and can influence the design. The same principles carry over to the design of complex new computers.

The second inspiration comes from an evaluation study of virtual memory reported in 1969 by David Sayre of the IBM Research Center at Yorktown [Sayr69]. Many prior attempts at evaluating the efficiency of virtual memory --

a new architecture in the 1960s – simply ran programs designed for manual overlays on a machine that handled overlays automatically. Those studies often led to the conclusion that the old programs would run slower on the new machine; but the conclusions were often disputed because they did not take account of the fact that users adapt to the new environment. So the real question – how efficient is virtual memory at handling programs designed by users who are aware of the virtual memory environment? – was unanswered until Sayre and his colleagues studied it. They demonstrated that programmers aware of program locality, the basic concept of virtual memory, created new versions of old programs that ran more efficiently on the new machine than the old programs ran on the old machine. They also demonstrated that the new architecture had a significant effect on lowering the cost of the programming process itself. Sayre's study stands almost alone in the annals of computing as a project that seriously attempted to assess the usability of the new architecture as well as its performance. Such studies have probably been rare because they are time consuming and difficult to plan. Yet they are essential.

The third inspiration for this study was recent. In 1983, Jack Dennis of MIT suggested that a workshop in which real users attempted to write programs for the data flow machine he was designing would be extremely valuable for two reasons. First, he felt it would allay fears that the machine was so radical that it would be difficult to program and use. Second, it would provide the designers with information about the limitations of the design while there was still time to alter it. We at RIACS agreed to host a workshop to evaluate Dennis's machine, called the MIT Static Data Flow Machine, in September 1984. Funding for the workshop was provided jointly by NASA Ames Research Center and DARPA. This workshop specifically addressed the suitability of the MIT machine for computational applications of interest to NASA Ames and DARPA [Adam85]. Our experience at organizing this workshop and evaluating the results has sharpened our appreciation for the methods of understanding how a new architecture interacts with its users.

2. Approach to Evaluation

The first step in any evaluation is careful determination of a set of metrics. The process of solving a problem by computer divides roughly into two parts: computation and programming. The evaluation criteria can similarly be divided into two broad categories. Metrics for computation are generally objectively measurable; they include the instantaneous speed of the hardware, the average speed of programs, queueing at various points, response times, and throughputs. Metrics for programming are generally more subjective; they include assessments of productivity and opinions about the usability of various features of the hardware, the languages, and the operating system. Metrics for computation are generally easier to obtain than metrics for programming. We advocate greater

emphasis on programming metrics.

2.1. Problem Solving Model

As software tools advance in sophistication, users will think of computers more as support systems for problem-solving than as mechanisms to write and run programs. Our approach focuses on the ability of a machine to support problem-solving in given disciplines. Therefore, we need to formulate our evaluation criteria with respect to a model of the process by which scientific problems are solved computationally.

Our model of the process of problem solving with a computer is depicted in Figure 1. The model treats a solution as a sequence of increasingly detailed algorithms. Each stage of the model is a transformation to a more detailed form, incorporating more knowledge about the approach and the architecture of the system on which the solution will be performed. The input to the highest stage is a description of the problem in natural and mathematical language. The output from the lowest stage is machine codes.

One of the purposes of such a model is to help determine the best stage at which to deviate from the previous solution process to construct a new process leading to machine codes for the new architecture. The model depicted in the figure is certainly not the most general, but it is a good starting point for a discussion.

The process of developing a computer solution to a problem begins with a requirement, usually called the problem statement, given typically as informal prose. We refer to this primitive representation of the solution as "stage zero."

Using knowledge, terminology, and concepts from the discipline, together with the language of mathematics, the scientist derives a precise statement of the solution. This statement may be a simple function or a complicated set of equations. We refer to the transformation from the informal solution to the precise formulation as "stage one."

In the next step, the solution mathematics are transformed into an abstract algorithm. This is a machine independent algorithm expressed in discrete mathematics, in logic, and in functional forms. At this stage selections are made for discrete approximations to continuous functions, numerical methodologies, general algorithmic approaches, general organization of the data, error control, recursion, and partitions into potentially parallel components. Numerical approximations, if necessary, are selected. These selections are influenced by the model of computation implemented by the target machine, but not yet by specific details of the machine's hardware. We refer to the transformation from the mathematical model to the abstract algorithm as "stage two."

Next, the abstract algorithm is transformed into a set of programs in real computer languages. It is possible that different components of the abstract

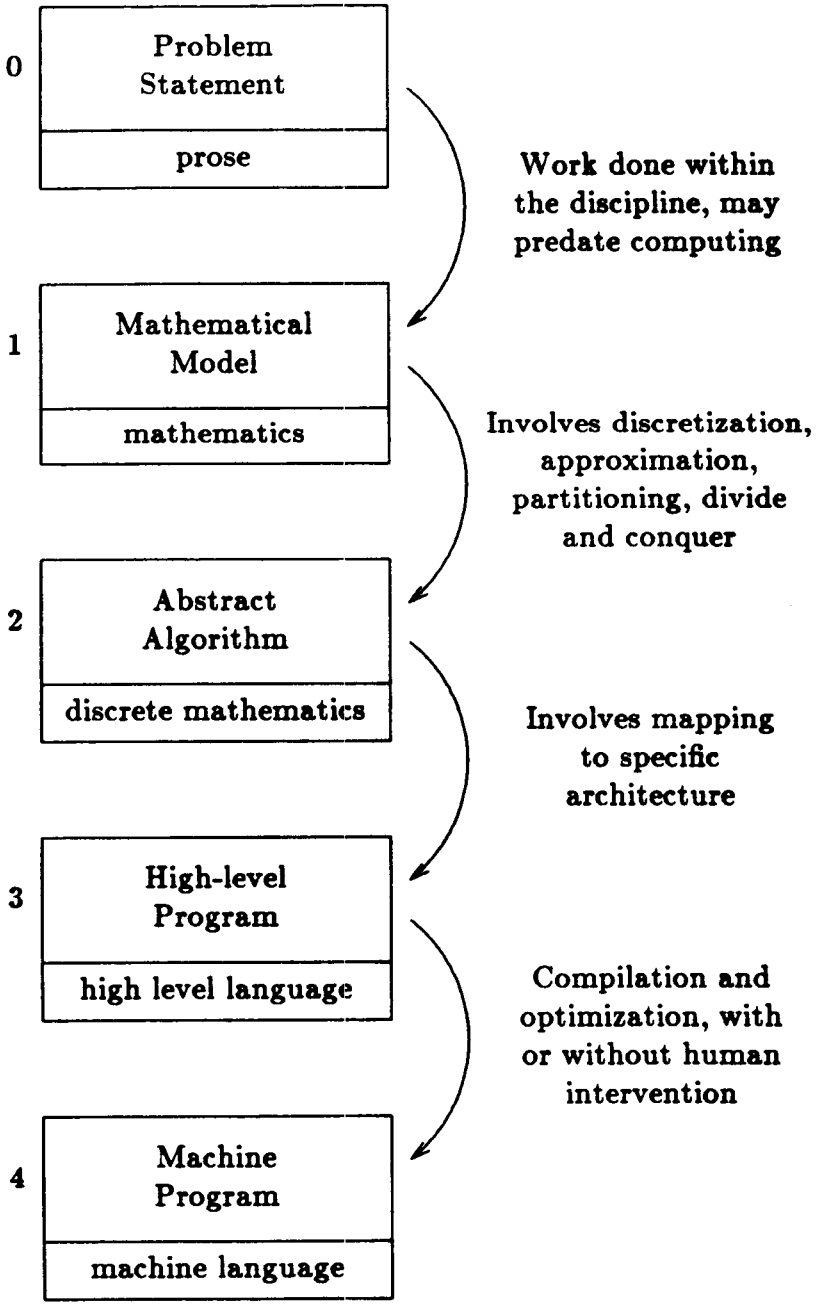

Figure 1. Steps in Scientific Problem Solving

algorithm require different languages for their best implementation; thus the applicable languages should be mutually compatible. For each component of the abstract algorithm, specific choices are made about factors that must be expressed in a programming language -- e.g., data representations, data flow, control flow, and precision. Specific characteristics of the machine will influence these choices -- e.g., word size, memory size, existence of vector instructions, and number of processors. We refer to the transformation from an abstract algorithm to a collection of concrete algorithms as "stage three."

The final step is translation of the concrete programs and their interconnections into machine code and linkages that can be interpreted by the underlying hardware and networks. The technology for these steps is well understood for conventional machines and is incorporated into compilers, linkers, networks, and operating systems. We refer to this transformation as "stage four."

There is an attractive strategy to adapt a given application on the current machine to a new computational model on a new machine. This is to lay out the stages of the problem-solving process that led to the current solution (stage four) and then backtrack only as far as needed to change the decisions that were influenced by the current architecture and computational model. Some examples:

1. The original program is in FORTRAN and is to be moved to a new machine. The original program can be recompiled using the new machine's FORTRAN compiler. This corresponds to backing up to stage three without changing any prior stage.
2. The original program is in FORTRAN and is to be moved to an APL machine. The scientist extracts the abstract algorithm from the FORTRAN program and reimplements it as an APL program, then uses the APL interpreter to solve the problem. This corresponds to backing up to stage two.
2. The original program is in FORTRAN and is to be moved from a VAX™ to a Cray-1. In this case, the scientist may need to change the strategy of the abstract algorithm to arrange the data and iteration patterns to take advantage of the vector pipelines on the Cray-1. This corresponds to backing up to stage one.

2.2. Evaluation Criteria

We noted above that the traditional criteria for evaluating performance are computation-oriented -- they evaluate measures of instantaneous speed of the hardware, average speed of programs, device utilizations, throughputs at selected points in the system, and response times. We also noted that another set of criteria has become important, namely those that are programming oriented -- they

VAX is a trademark of Digital Equipment Corporation.

evaluate the utility of a language, time to construct a working program, ease of debugging, modularity and maintainability of code, productivity of programmers, cost of a solution, and opinions about system usability.

Actually, these two sets of criteria are closely related. In the model of scientific problem solving, the programming criteria seek to evaluate the four stages. The computational criteria focus on the results of the fourth stage only. That computational criteria are more common than programming criteria is simply an indication that in the past most changes in architecture did not change the sequential process model on which most programs are based and, hence, did not require changes to the process above stage three.

The amount of effort one is willing to invest will constrain the extent to which the stages of the problem solving process can be evaluated. Generally, the less one spends, the more the process will focus on the higher numbered stages only. There are roughly four levels of effort.

1. **Raw performance evaluation.** The burst performance of a machine, expressed in operations per second, may be computed from knowledge of the clock rates and from the speeds of important components such as memory, register, and bus. This provides an absolute upper bound on the speed at which operations can be executed. The advantage of this level of evaluation is that it permits machines to be compared without actually running programs. The disadvantage is that few real programs run at average speeds approaching the machine's instantaneous speed. The Cray X-MP, for example, is rated at 650 MFLOPS; most FORTRAN codes realize only about 100 MFLOPS average and a few hand-optimized codes may attain 150 MFLOPS. The reason is that the sequential parts of the computation dominate and cannot be made faster by vector pipelines.

2. **Small function programming.** At this level, a few common library functions, such as matrix multiply or FFT, are carefully coded in machine language, then tested and tuned by hand. The objective is maximum speed on the target machine. The finely tuned functions of one machine are compared with their finely tuned counterparts on another machine. The advantage of this level of evaluation is that only a few, frequently used programs need be rewritten and recompiled; it is a good way of quickly "stress-testing" a new machine. This can lead to a higher degree of confidence in comparisons than an evaluation at the previous level. The disadvantage is that the selected programs are only a small portion of real programs, hence, inferences about real programs remain weak.

3. **Benchmark programming.** At this level, a substantial software package, such as LINPACK or Livermore LOOPS, is reprogrammed for the new machine. It is run and timed for a standard input. Machines are evaluated by comparing the results. This approach is an

expansion of the concept underlying the previous level. The advantage is that inferences about machines are more likely to be observed in practice because the benchmarks can be constructed to accurately reflect the statistics of the actual workload on the system. The disadvantage is that the approach gives no information about the effort of programming experienced by users of the machines being compared.

4. **Complete application programming.** The machine is evaluated by programming, running, and observing entire end-user applications. The programming is usually best performed by experts in the domain of the application. An entire programming environment consisting of editors, compilers, linkers, and debuggers is required to perform these tests. The advantage is that programming effort and the utility of software tools can be directly evaluated in any given discipline. The disadvantage is that these evaluations presuppose a good repertoire of utility software and require a good deal of time to prepare and carry out.

The work at each of the four levels is often done by a different group of people. For example, raw performance and small function programming is most often done by the architects of the new system. This group best understands the lowest levels of the machine and how to achieve the greatest speed from the design. Benchmark programming is often performed by a group representing end users; the users themselves are not actually involved. Application-level evaluation is performed with direct cooperation of the users. Typically there is little communication between the architects (who test at the first two levels) and the users or their representatives (who test at the last two levels). This means that limitations discovered by the users must often be tolerated by the users rather than repaired by the architects.

To overcome the communications gap between users and architects, we advocate building evaluations at the fourth level into the process of building a machine. This maximizes the opportunity for discovered limitations to influence the design in a positive way. We believe this is especially important with the new generation of concurrent architectures, where the amount of experience is small, the project lead time is large, and our intuitions are undeveloped.

3. Methodology of an Actual Evaluation

In the fall of 1983, Jack Dennis suggested that RIACS host a workshop in which potential users of parallel computation could work with him in an attempt to program real problems on the MIT Static Data Flow Machine. Most of the architecture of the machine had been specified but some parts were incomplete; a programming language, VAL, a compiler, and an interpreter had been built. The only VAL programs that existed at that time were written by the machine's

architects and students, but not by users in actual disciplines.

The purposes of the workshop would be several: a) determine whether users would find that massively parallel computations were easy to program in the data flow language VAL; b) obtain preliminary evaluations of the speed at which key algorithms might run on the data flow machine; c) get feedback to influence remaining design decisions; and d) give NASA and DARPA a preliminary evaluation of the ability of data flow architectures to provide needed computational power in key disciplines. By spring of 1984, we at RIACS had agreed on a plan for a two-week workshop to be held in September 1984, and had obtained funding for this effort from both NASA and DARPA. The workshop was held as planned. It consisted of seven teams, staffed by NASA and RIACS scientists, each expert in a particular discipline. Jack Dennis and two colleagues from MIT participated; they lectured on the principles of data flow programming and gave detailed advice to the individual team members. RIACS supplied offices, a computer system, and administrative support; they analyzed the data and issued a final report in March 1985. All parties judged that the workshop succeeded in its purposes [Adam85].

3.1. Planning Maxims

In planning for the workshop, we applied the general principles stated earlier. For the context of a two-week study, we refined the principles into a series of maxims that guided the details of administering the workshop and evaluating the results. The maxims are stated and elaborated below.

1. Plan ahead. It goes almost without saying that a two-week workshop with many participants, a workshop that sought to learn much in a short period, needed careful planning. Funding, scheduling, location, and staffing were all prepared well in advance.

2. Bring in the architects. If evaluation is to be an integral part of the development of a new computer, the designers must participate. Jack Dennis and two colleagues from the design group of the MIT Static Data Flow Machine came to RIACS for the workshop. Their role was to provide instruction on the MIT machine and VAL and expertise for consultation on data flow computation.

3. Bring in scientist volunteers and organize them into teams. We obtained 11 volunteers. They were organized into seven teams representing seven application areas of interest to the study sponsors, the MIT contingent, and RIACS. Because it was important that the participants be unbiased with regard to data flow computation, we asked for volunteers who had an open mind to new computational schemata and who also had no prior experience with data flow programming. Because it was important that the participants give expert assessments of the usability of the MIT design, we asked for volunteers who were active in creating, refining, and evolving production code used in their disciplines. We asked each team to bring a working supercomputer code to the study

for comparison with the data flow codes they would create.

4. Schedule a formal workshop. To assess the usability of the new architecture, we needed to observe the scientists directly as they worked and interacted over a protracted period, in a setting conducive for research. Two weeks was about the maximum amount of release time we could arrange for the volunteers. The two-week duration limited the size of the problem that could be studied by each team. It was nonetheless sufficient to allow the seven teams chosen to consider a kernel problem in their disciplines. The workshop format has the additional benefit of isolating the participants from their daily work -- free from distractions of telephones, inquisitive coworkers, and managers. (We did schedule the daily starting time late enough to allow participants to take care of urgent daily tasks before coming to the workshop.)

5. Provide a familiar programming environment. Most of the workshop participants were not familiar with the UNIX™ operating system used at RIACS or with its editors. The MIT staff were familiar with MULTICS and the Ames scientists with VAX/VMS™. To accommodate these diverse backgrounds, we borrowed several DEC VT100 terminals for the scientists and provided an emulator of EDT (the VMS editor) within the UNIX emacs editor. This was extremely successful. We needed to provide only a single sheet of system documentation telling how to log on and off the system, how to invoke the editor, and how to use the VAL translator and run-time system.

6. Plan an agenda of lecture and lab. We planned a schedule in which the first few days would be mostly lectures by the MIT staff on data flow concepts, machine architecture, and the VAL language. Lectures were alternated with terminal sessions in which the participants experimented with VAL programs. The remaining days were scheduled mostly for lab work with additional lectures given on special topics as required. The MIT staff were available throughout the lab sessions to serve as consultants on data flow programming. All documentation on the MIT Static Data Flow Machine was made available in personal copies for each participant. This schedule worked well. By the third day, the participants were ready to begin programming their selected problems in VAL.

7. Automate data collection where possible. We carefully considered what metrics would be suitable for this workshop and how to collect the data. Because the MIT architecture was not completely specified, it was not possible to measure VAL programs in execution; accordingly, we made no attempt to instrument the intermediate-language interpreter. Because we did not know in advance what metrics would contain the most information, we recorded everything: the input and output of each compiler and interpreter run by each participant. From this we were able to extract measures such as number of compilations, number of errors, compile times, interpreter times, and the like. We

---

UNIX is a trademark of AT&T Information Systems.

decided to err on the side of collecting too much data because, when the workshop was over, nothing more could be gathered.

To obtain subjective information from the participants, we used questionnaires, diaries, and reports. We constructed a questionnaire for the participants to complete at the end of the workshop; we solicited their opinions and comments on every aspect of the study. We asked each participant to keep a file containing a daily dairy of impressions. The RIACS and MIT staffs kept diaries of their impressions and comments received during direct interactions with the participants. We asked each team to submit a written report on their project within two weeks after the workshop ended.

The responses to the questionnaire constituted the most illuminating feedback from participants to the system designers. These comments revealed that at least three aspects of the architecture needed further thought (the ability of the hardware to scale over a wide range of processing elements, the method of matrix data storage, and the I/O system bandwidth).

8. Keep data on-line. We asked participants to keep notes containing impressions of the day's activities on-line. We did not insist strongly on this and, unfortunately, there was too much variation among participants. If we were to do this again, we would schedule a short block of time at the end of each day during which participants would be required to make entries in their notes. The questionnaires were filled out by hand but put on-line by a secretary. The final reports were written on-line using the emulated editor. It significantly simplified data analysis and report generation to have all this material available in this way.

3.2. Observations on the Workshop

The most important observation was that workshop met its four goals. a) End users assessed the ease of programming in the data flow language VAL. b) The final reports did contain preliminary evaluations of the speed at which key algorithms might run on a data flow machine. c) Several important limitations of the design were uncovered by the users and the architects set out to correct them. d) NASA and DARPA received a preliminary evaluation of the ability of data flow architectures to provide needed computational power in key disciplines. We conclude that the general organizational principles discussed above are achievable in practice.

We were surprised and pleased by the sustained effort made by the volunteer participants. The initial two weeks of activity led to useful preliminary results. We concluded that a longer study – three weeks, for example – would not have significantly increased the utility of the preliminary results. Most participants continued to work on their projects long after the workshop

VMS is a trademark of Digital Equipment Corporation.

ended, turning their final reports into papers submitted for publication. Some participants suggested that the two weeks be separated by a one or two week interval during which they could think more deeply about data flow programming and feel more prepared for the second week. Based, however, on the amount that was accomplished and the fact that participants completed their work in spare time after the formal end of the workshop, we are not convinced that this change would have materially affected the results. We conclude that a two-contiguous-week study was adequate for the stated purposes.

The participants appreciated the effort to provide terminals and editors. The office space set aside for the study made it easy for the MIT participants to sit down with individual scientists for extended interaction without interruption and without risk of breaking the concentration of other participants. We conclude that the physical environment in which the workshop is conducted is an important factor in the successful outcome.

Our study involved a model of computation different from that in which any of the applications codes had been developed. Thus we watched carefully to see how far back in the problem solving-process scientists had to retreat before being able to construct a data flow solution. We observed that most participants did not back up past "stage two," the abstract algorithm stage. They kept the same mathematical model and abstract algorithm as were used to construct the existing FORTRAN code. When questioned about this, most argued that the abstract algorithm exhibited a high degree of parallelism, sufficient to absorb the available processing capacity of the MIT machine specified for the study and, hence, further retreat was not warranted in the short time available. We concluded that for the purposes of our study, these disciplines were justified in considering no further retreat in the problem-solving process. This conclusion may not carry over to other disciplines, or even more difficult problems in the same disciplines. This aspect of a study is worth monitoring closely.

The types of information collected were useful and revealing. It may be useful when conducting larger studies to employ an expert consultant to assist in designing the questionnaires and other methods of subjective data collection. This would help reduce bias and improve the ability to compare the results of this study with later studies of the same machine or different machines.

Our study was time consuming and laborious. Its budget of $20,000 significantly understates the actual effort expended for several reasons:

1. None of the participants' time was charged to the effort. The MIT staff time was covered by their own research grants. NASA staff time was donated to the project by their managers granting release time. The 17 people directly involved in the study collectively provided 28 person-weeks of effort during the workshop proper, and may have donated a like amount of time for follow-up work. Thus the sponsors provided the equivalent of more than one person-year of effort that did not appear in the budget.

2. The authors significantly underestimated the effort to analyze the data, consult with the team participants about their reports, and prepare the final report. Another four man-months of effort was expended to this end.

References

[Adam85]

Adams, G. B., R. L. Brown, and P. J. Denning, "Report on an Evaluation Study of Data Flow Computation," RIACS TR 85.2, Research Institute for Advanced Computer Science (Apr 1985).

[Sayr69]

Sayre, D., "Is Automatic "Folding" of Programs Efficient Enough to Displace Manual?," *CACM* **12**(12), pp. 656-660 (Dec 1969).